Reducing an attack surface of an operating system

Ville Valkonen

Abstract:

Certain security choices done on the operating system level can mitigate harm done by an malicious attacker or a program. The main focus in the thesis is on open source operating systems.

Keywords: Software security, system security
CR-classes: D.4.5, D.4.6

1 Introduction

This thesis examines different proactive security methods that can be integrated into operating systems and therefore reduce a damage made by malicious activity. Some of the methods are easier to adopt in general programming guidelines while others are more dependent on the environment. The main focus of this thesis is on open source operating systems.

Difference between the high secure systems and non-secure system can be described on the following way: non-secure systems concentrate on implementing certain functionality while secure system's main goal is to achieve functionality in a safe manner. Designing a secure system is an arduous process and difficult to get correct. Basically, a system should *fail safely*, should run the *least privilege* and should not be too complicated to understand (*KISS*). These topics are discussed in greater detail below. As Erik Poll [10] has stated "The attacker only has to get lucky once, the defender has to get it right all the time".

2 Preliminares

This chapter focuses on different technologies and methods that are widely adopted by many known security environments, programs and security focused operating systems. Hence, systems that lack of exhaustive testing process or are in a prototype state are excluded from this thesis. This chapter also presents shortages that current approaches might have. By all means this thesis tries to give an general understanding of an operating system vulnerabilities that can be pursued – but does not try to cover all possible fields or methods. There are fields regarding concurrent processing, networking and input/output processing that are omitted in this thesis, though some of them might be mentioned briefly.

At the beginning we define what is a software flaw and what proactive functionality can be done to prevent or reduce the damage. The software flaw is an unexpected operation of a program. It can expose system to a great danger, leak confidential information or crash the whole system. On some cases, especially in network related environments, the software flaw can pose huge number of computers to a danger. If it is possible to gain more information from the system than it is designed to offer, or when the system is set up to work in a way it was not intended, it can be seen as a vulnerability.

There are several different types of vulnerabilities. One can measure vulnerabilities according to severity, while one can measure its level of criticality. Security companies rank vulnerabilities different way but certain parts of the rankings are the same. One of the unarguable ranking category is its type of a domain, *a local exploit* and *a remote exploit.* The local exploit demands that attacker must have a local access into the system. The local access can be physical or non physical (an user account in the machine). The remote exploit instead of does not have this demand and can be therefore seen more dangerous. Remote exploits are considered more risky since there is no need to have the account in the machine. Hence, the latter can be done via Internet.

Communication in the Internet is based on a packet data, thus a client has to negotiate connection to a server (in normal cases). Negotiating the connection defines a common language for both parties, this is better known as *a communication protocol.* Reporting the *user agent* information is part of the communication process. Without including the exceptions user agent information usually includes a program version. This information defines whether the server and the client can communicate together, i.e. the client has apt version of a program. By examining the communication information malicious user can gain enough information to pursue the functional attack against the service.

Evans and Larochelle demonstrated [3, pp. 8-9] the use of the *splint* [14] tool against *wu-ftpd* [21] daemon. At time their paper was released wu-ftpd was a widely used ftp daemon [1]. Splint is *a statistical code analyzer* that detects certain type of weakness from the program by analyzing its source code. Detecting a basic buffer overflow and misuse of functions is a part of splint's reportoire. By examining the tool result they found bugs that were not found before, as well as bugs that were already known by the vendor. Although they used statistical code analyzer against the source code, it is possible to "practice" exploit scenario before fulfilling it. Furthermore, as the analyzed program was interacting with other computers, it is possible to use studied information to arbitrary exploit daemons around the Internet.

In addition, it is possible to use code annotation where the especially crafted syntax is inserted within the code. This functionality needs support from the compiler. Annotations makes possible to add more fine-grained checks into the code and get better understanding of the program. Anybody can review open source software and review the code, use tools like splint to analyze the code and generate an exploit. This can be seen as a bad and a good thing. Bad, as people can read the code and spot flaws more easily. Good, because code gets likely more reviews and bugs get fixed by anyone that has certain understanding. This certainly needs interaction with the community.

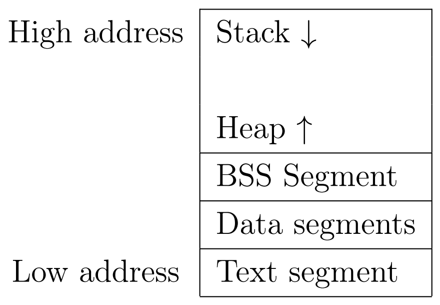
Unless explicitly mentioned, all examples in this thesis are related to a Unix based operating system.

2.1 Memory

Different kinds of attack methods that can be pursued via memory are reviewed in this section. Moreover, in order to gain higher understanding

[1]In Unix, daemon is a background process

regarding the protection methods.



Example 2.1. Memory layout of the Unix process. [18, p. 2]

Since it is easier to understand the process memory layout by having a concrete real life example (see examples 2.1 and 2.2), we use the same code that John Wilander and Mariam Kamkar used [18, p. 3] to describe the process.

Example 2.2.

```
static int GLOBAL_CONST = 1;        // Data segment
static int global_var;              // BSS segment

// argc & argv on stack, local
int main(argc **argv[]) {
    int local_dynamic_var;          // Stack
    static int local_static_var;    // BSS segment
    int *buf_ptr=(int *)malloc(32); // Heap
}
```

By examining the source code we can conduct that arguments, constants and variables are stored in the different places in the stack (see example 2.3). For instance, statically declared integer variables are stored in the BSS stack. Machine code is the deviant in the memory layout that cannot be observed from the source code example. It is stored into the text segment.

Every time a function call occurs, the stack grows by one (will be the top element in this implementation). Each of these calls include local variables, old base pointer, return address and arguments.

Lower address	
	Local variables
	Old base pointer
	Return address
	Data segments
Higher address	Arguments

Example 2.3. A stack frame. [18, p. 3]

Various attack methods concentrate to alter the stack frame such a way that attacker can execute arbitrary code and potentially gain higher privileges.

2.1.1 Buffer overflow

Over decades buffer overflow attacks have been populating the software security flaw top-lists [2]. Buffer overflow occurs when a process writes to the memory address out of the allocated area. Buffer overflow enables code injection exploitation through the current process. This is known as an arbitrary code execution. Demanded attacker can gain himself or herself the same privileges that the process has.

linenro	code
1	char mymsg[2] = "hi";
2	strcpy(mymsg, "hello world!");
3	printf("%s\n", mymsg);

Example 2.4. Simple buffer overflow example in C language (only the relevant parts) is shown in the code fragment.

In the example 2.4, line 1 contains the first flaw: length of the *mymsg* string is 2 but strings should be terminated by the *NUL-terminating character* \0. NUL-termination is crucial since many functions in C depends on that. Without termination character function does not know when to stop. It continues until some memory address includes one.

In the line 2, *strcpy* function is used for copying a string *hello world!* into the *mymsg* variable. Since the size of the *hi* is smaller than the copied string *hello world!* buffer overflow occurs. This makes the program indeterministic.

Although one certain type of overflow was examined in the example 2.4 several variations of buffer overflow exist. Theo de Raadt [13] lists the following variations: stack overflows, data segment overflows, GOT/PLT overwrite, jumping to data that attacker can execute, 4 byte modification possibility and 4 byte read possibility.

2.2 Access control methods

Access control methods limit the access of a certain resource. They can be also used to limit visibility. However, access control must not be mixed with authentication. Access control checks whether a user has permission to complete certain activity, in contrast to authentication which identifies user [22]. Several access control methods exists for different purposes. Some offer greater granularity and are therefore more flexible. This means better control for administrators.

Discretionary access control (DAC) is the simplest access control mechanism that is discussed in this thesis. It has gained popularity among Unix operating systems since it has fairly clear structure that follows the KISS design principle. Nowadays many modern operating systems have moved towards more fine-grained access control, such as an access control list (ACL) and a mandatory access control list (MAC).

The DAC has only two main features, ownerships and delegations. A user who create an object is also an owner of the object. Owners have all the rights for the object and owner can delegate certain rights to the users.

Authorization states can be expressed as an access control matrix which includes two properties: subjects and objects. Hence, columns presents objects and rows presents subjects.

Definition 2.1. If the access matrix is A, subjects is s and objects o, then $A[s, o]$. These properties together form a triplet (S, O, P) [5].

	File 1	**File 2**	**Program**
Alice	own read write		
Bob	read	read write	execute
Charlie		read	own read write

Example 2.5. DAC access matrix.

Example 2.5 depicts access control matrix in Unix system. Despite the fact that DAC is pretty flexible for regular use, it does not control how information is forwarded. Therefore, a trojan horse attacks are completely viable. In distributed trust management lecture notes *** , Daniel Trivellato made an example where Alice grants Bob with reading permission but does not want to Charlie to be able to read the information (see example 2.5). Bob can leak information to Charlie without even knowing it. Attack is achievable by using the trojan horse technique which was shown in the lectures notes with greater detail (see pp. 11).

ACL expands DAC that one file can have have different access control rules. With ACL it is possible to attach rules like *Allow Bob read and write the File1, Charlie to read and Alice to write.* [4]

lattice based access methods, slides 1.

3 Coping methods and discussion

3.1 Design principles

Several design principles that are known to improve error handling, increase the safety and tune the controlling abilities should be used in a security oriented programming. Obeying these principles does not guarantee flawless programs but help to cope with the problem. These principles should be usually perform at the beginning of the design process, since it is easier to design functionality around them. It is viable to adopt the methods later on but usually it means vast changes into the code.

3.1.1 KISS

KISS is an acronym for Keep It Simple, Stupid! Principle encourages simplicity over perfection *** Ritchie and Thompson lähde tähän. According the principle, more lines of code implies more obscure structures and therefore makes it harder to understand. Updating the code comes harder and implementing a new functionality can be challenging. In general, more lines of code increases bug count and poses more (security) flaws. Moreover, program becomes less deterministic and ambiguous at many level. Original Unix was designed to adhere KISS principle, therefore one tool is designed to accomplish one operation.

3.1.2 Whitelists

Whitelist is opposite of blacklist. Instead of denying forbidden functionality whitelist allows certain functionality or operation. If the capability needs are not met it is denied by default.

An excellent usage of whitelisting is presented in the program *TCP wrappers* [15]. TCP wrapper is used for limiting hosts access to services. In addition to whitelisting, it allows blacklisting as well.

3.1.3 Least privilege

When running a process it should not run higher privileges than is needed. Therefore, if possible, process should drop all extra privileges after the required high privilege job has been completed.

Example 3.1. HTTP daemon needs to bind to port 80 (HTTP traffic) in order to be able to serve web pages for web browsers. Ports that are equal to or lower than 1023 are called as *low ports*. In order to bind socket to these ports, *super user* privilege [2] is required to accomplish the task. After completion the port binding daemon should drop all extra privileges that are no longer needed.

[2] *root* user has the highest privilege. Also, ID number 0 refers to root.

However, as Provos et al. [11, p. 2] state, this does not guarantee safeness. Certain type of flaw in a daemon process can still leak higher privileges to a malicious attacker. As a countermeasure, Provos et al. propose *privilege separation* as an addition to the least privilege. For more details about privilege separation on below.

3.1.4 Fail securely

When malfunction happens it must be taken care on appropriate manner. If one level of security fails there should be another level to mitigate malfunctioning (*defence in depth*). For instance, a computer that is linked to the Internet should not solely rely on firewall [10, p. 9]. It is even more crucial when the computer runs services.

```
You have an error in your SQL syntax; check the manual that corresponds to
your MySQL server version for the right syntax to use near '

and afdeling.provincie in (0,3) order by afdeling.sort' at line 3

The error occurred in D:\websites\kicker\content\Uitslagen\index.cfm: line 32

30 : select distinct(afdeling.afdelingid) as afdelingid, afdeling.afdeling
     as afdeling
31 : from reeks left join afdeling on (reeks.afdeling =
     afdeling.afdelingid)
32 : where seizoen = #qGetseizoen.maxseizoen# and afdeling.provincie in
     (0,#regio#) order by afdeling.sort;
33 : </cfquery>
34 : <cfoutput>

SQLSTATE   42000
SQL  select distinct(afdeling.afdelingid) as afdelingid,
      afdeling.afdeling as afdeling from reeks left join afdeling on
      (reeks.afdeling = afdeling.afdelingid) where seizoen = and
      afdeling.provincie in (0,3) order by afdeling.sort;

VENDORERRORCODE 1064
DATASOURCE      kickersql
}
```

Example 3.2. An example presents an inappropriate way to handle errors.

The example 3.2 leaks several important information to an attacker. By knowing the database and table names it is possible to try database related attacks like SQL-injection to gain higher level access into the system. The path name reveals used operating system which is in this case Windows. By examining the stack trace (see appendix) it reveals used programs and platforms (MySQL and ColdFusion). The previous information might be enough for the malicious attacker to penetrate into the system and/or crash the site.

Cope to the problem in here is to appropriate handling of debug and system messages and writing errors to a log file instead showing them to user [10, pp. 35-41].

The previous case happened to the author in the particular site. After reporting misconfiguration to the site administrator it had no effect nor they replied to an email. An good example how low priority security has for many administrators.

3.2 Compiler and language based safety methods

Machine language is the lowest language that computer “understands”. It is hard to read and understand for human since it does not obey structure of a human language nor have similar lexical or syntactical expressions. *A symbolic machine language* remedies this shortage. By using the symbolic machine language multiplication operating is expressed as MUL [17, pp. 1-2]. Syntactically it is closer to *natural language*, though it the differs lexically a lot.

Computer languages can be split into two different categories, *low level languages* and *high level languages.* The previous paragraph discussed about the machine language and the symbolic machine language. Both are part of the low level language category. While these both languages are formal languages, these also share a strict grammar. Vice versa, languages that human use for communications are more subtle, ambiguous, loosely defined and might vary syntactically a lot. As low level languages have a simple instruction set, it is hard to accomplish highly abstract tasks like implementing a B-Tree data structure. For this purpose there are higher level languages that more subtle as compared on its predecessors, low level languages.

In a design process one should carefully choose the implementation language. The language that guarantees *type safeness*, *integrated sandboxing* and is pointer free, should be always preferred. By using the language that implements type safeness it is impossible to mix strings and integers on the following way: $2 + "\#"$.

Misuse possibility exists if the language offers memory operates via pointers. Therefore pointer free (*memory safe*) languages should be preferred. Occasionally program has to be implemented in a language that does not fulfill these security needs. In this case, functions that are boundary aware should be preferred and function return values should be verified in the case of errors. Also, function return values should be always checked against errors if the language does not implement exception handling itself.

In the low level language program attacker can craft an exploit that is highly dependent on machine architecture instructions and bypass higher level language limits. Albeit this is an interesting topic, the main focus in this study is in the higher level languages.

Majority of the open source operating systems are implemented in C language because it has support for symbolic low level language. It is possible to write hardware drivers and make certain functions faster in lower level implementations. C language's main strength is in its speed. The language has many functions that gain speed by omitting boundary checks and by accessing memory directly via pointers. Although the speed assumption is true in general cases, Miller and de Raadt pointed out [9, p. 2] a safe function implementation that does not have critical speed regression.

One of the functions that lacks off the boundary check is *strcat*. strcat is used for string concatenation. A manual page for *strncat* regarding the Open Group Base Specification [16] is Figure 3.2 (only the relevant parts are included):

```
SYNOPSIS
    #include <string.h>

    char *strncat(char *restrict s1, const char *restrict s2, size_t n);

DESCRIPTION
    The strncat() function shall append not more than n bytes (a null byte and
    bytes that follow it are not appended) from the array pointed to by s2 to
    the end of the string pointed to by s1. The initial byte of s2 overwrites
    the null byte at the end of s1. A terminating null byte is always appended
    to the result. If copying takes place between objects that overlap, the
    behavior is undefined.
```

Definition 3.1. Unix manual page of the strncat

Prototype of the *strncat* takes exactly three arguments: $s1$ (destination), $s2$ (source) and n (size). If a source buffer size is greater than or equal to a destination buffer, NUL-terminating string is omitted. To use *strncat* safely, Miller and de Raadt [9, pp. 1-2] encourage to always copy size of $n - 1$ and NUL-terminate string by hand, although in some rare cases the previous operation is exaggerated.

Miller and de Raadt also disclose the misuse of the *strcnat* [9, p. 2]. Particularly the size parameter is often thought to be the size of the destination buffer. Quoting Miller and de Raadt, they use the following formalization to clear this misconception:

> *Most importantly, this is not the size of the destination string itself, rather it is the amount of space available.*

As their last concern Miller and de Raadt [9, p. 3] pointed out that the performance regression regarding the speed made by the boundary check is as it uses more CPU cycle times. During the time paper was released (1996) computers had significantly lower performance. Today when CPUs have more than doubled the calculation power and can do two operations in a single instruction, this claim is considered outdated.

3.2.1 Automatic memory management

There are two types of variables: local and global variables. The local variable has visibility in a function block. The global variable can be accessed anywhere from the same file. Many languages have adopted a syntax where *scope* is defined as an area between the curly brackets, { }. Example 3.3 code fragment depicts the previous situation.

```
variable A   // Global variable

function() {
        variable B   // Local variable
}

main() {
        print(A)   // Since A is global, this works
        print(B)   // Does not work, since B is local under function()
}
```

Example 3.3. Code fragment regarding the local and global variables.

When a program returns from a function any of the locally reserved variables cannot be accessed anymore (excluding global variables). Age of the variable is consequently as long as the program execution stays in the same scope.

Exceptions exist depending on the chosen programming language. C, for example, has exception if a local variable is declared as a pointer. Moreover, if the pointer refers to a dynamically allocated memory it should be freed before leaving the function. It is also possible to return the pointer so dynamically allocated memory can be accessed outside the function.

However, the previous approach has certain drawbacks. If one forgets to release the dynamically allocated memory in the end of the function, memory stays allocated and reference to it is lost if the pointer is not returned. In the case the allocated information is confidential there is possibility to leak a confidential information. Next time when the memory allocation occurs and an attacker is determined, it is possible to allocate the same chunk of the memory. In certain languages that does not provide automatic memory handling, memory can be allocated without first emptying the memory area. One of these languages is C and to be specific, its function *malloc* can be used for the purpose. With certain knowledge the attacker can alter the previously used memory block and read the confidential information. One possible defend against this kind of attacks is to patch malloc to force erase allocated memory blocks. Such functionality is provided at least in OpenBSD malloc [7] (J and Z switches). Other approach would be to switch to a memory safe language.

Garbage collection takes care that all dynamically loaded chunks are freed afterwards and the previously mentioned leak cannot happen. Without garbage collection every dynamically allocated memory block must be released by the user. Omitting the deallocation operation makes it possible to have a memory leak.

3.3 Protecting the memory

There are several ways to protect the memory from the leaks and the buffer overflows. The main focus is in the ProPolice in this topic as it prevents different kinds of attacks as proven by Wilander and Kamkar [18, 13-14].

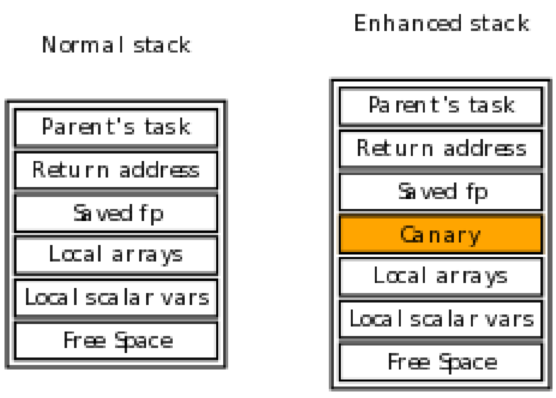


Figure 1: Structure of a computer memory stack.

In figure 1 shows an enhanced process memory layout that resides in a computer memory. Technique is known as ProPolice [6] and it protects against the *stack smashing attacks.* Protection works by inserting an extra information, canary (sometimes called as a cookie), into the stack. Canary is basically a random value. The canary is inserted into the each function call at the compile time [1, pp. 10-12]. If the canary has been changed program terminates.

Furthermore, ProPolice reorders the stack in a way that the flags and the pointers are placed lower in the stack. If overflow occurs, it likely first overwrites the canary. It makes harder to overwrite flags and pointers since the canary changes are noticed. ProPolice has moderately low regression to performance since it is only about 1.3%.

Even the compiler would use canaries to protect against stack smashing problem, there are still possibility to pursue the attack. To make buffer overflow attacks even harder to gain any benefit, OpenBSD operating system [19] introduces the following methods [1, p. 16]. Randomizing *ld.so* [3] memory load locations and orders, as well as randomizing *mmap* and *malloc* calls.

Other widely used approach against buffer overflow attacks is to use NX-bit. Abbreviation NX-bit stands for Non Execute Bit. It works by not to allow have write and execute right in the memory stack simultaneously. Its importance have been seen so crucial that modern processor manufacturers have implemented in a hardware level. Although it is widely used and works in general cases, it can be circumvented (proof of concept attack [8]).

3.4 Access control methods

Many different access control methods exist and these can be tedious to find appropriate use for each application. It should be clear for what purposes access control are going to be used. Some environments work perfectly with DAC, whereas other environments demands more fine-grained approach and control for the information flow. On the latter cases role based access control (RBAC) methods like MAC can be considered.

Some of the methods are too complicated to use consistently. ACLs are great example of ambiguous functionality where capability setting is not

[3]Run-time link-editor

straightforward. Files that have many distinct rules come quickly unmaintainable.

An another example is from the Windows and the Macintosh OS X world where MAC restrictions (on by default) displeases many regular users by not letting complete their tasks. After downloading a file from the Internet a dialog asks whether program should be executed. More similar pop ups will follow and screen is filled up with questions – just to complete a simple task. This leads to the situation where users turn off the MAC based access control and are again vulnerable to malicious applications.

3.4.1 Sandboxes

Sandboxing is an important factor when isolating running services or processes. Well designed sandbox implementation mitigates compromising the complete system in the case of malfunction of a program.

3.4.2 Chroot

The chroot (*change root*) changes a given directory visibility. If a directory $/home/user1$ is influenced by chroot, it points to the /, which equal for the root directory.

There are few drawbacks when chrooting a process. All the needed runtime dependencies must be copied inside the chrooted directory. Since $/home/user1$ is now seen as /, file system hierarchy must be imitated inside the chrooted environment. The referrers to an object that resides outside the chroot does not work. For instance, soft links pointing out of the chroot are superfluous. If the process must run with root privileges, chroot renders useless [12], since root can always escape from the chroot environment.

Example 3.4. If the operating system updates *libc* and the chrooted program is written in C-language, all the dependent libraries must be updated manually to correspond non chroot system library versions. This step can be omitted if system libc update in chroot is neglected. Previous method is righteous if system is not security critical or the program does not have any vital operations. Program *ldd* (list dynamic object dependencies) is a helpful tool when replicating the program into the chroot environment.

3.4.3 Privilege separation

In a basic non-secure environment clients interacts directly with the privileged process. This poses system to a danger since malfunction can leak higher privileges or grant access to confidential information. Niels et. al [11] propose isolation between the interacting clients and the server processes. By following way clients interact with non privileged process that have been forked from the privileged parent process. Privileged parent process can be needed for crucial tasks, like binding the listening address in low ports [4]. Splitting the processes makes program more resilient against attacks since malfunction only affects to the forked children. A mock-up in the figure 2 shows how communication is made when a client from the Internet requests a HTTP-page.

It is not always possible to drop higher privileges and therefore Niels et. al [11] integrated privilege separation into the *OpenSSH server* [20].

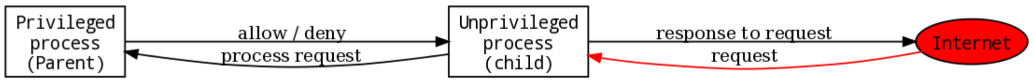


Figure 2: Communication negotiation between the Internet client and the service that have been privilege separated.

4 Conclusion

Completely secure systems do not exist. Users are and will always be the weakest link what comes to the computer security. Users write the software, develop the hardware and use the computers - appropriate or inappropriate way. Well designed security environment does not guarantee protection against the all plausible flaws that exists or will exist but makes system penetration and exploitation harder. Even the safest system cannot be left without updates after the initial setup has been completed on a safe manner. Security is race against the time where the only hope is a incompetence of a malicious attacker.

Many of the introduced methods offer safer way to execute third party programs, hide classified information, sustain integrity or make service fail on a

[4] $Lowport < 1024$

safe manner.

As the conclusion to the language based security, several badly written application programming interfaces (API) exists and are used daily. Also, as proven on above, many of them are ambiguous and hard to use consistently. A leap in a security is to substitute all unsafe functions with the boundary aware functions and use APIs that have consistent interface.

If the security is the main concern, performance regression is inevitable since values must be checked against the different safety limits. Only the magnitude of regression varies.

References

[1] Advances in OpenBSD. <`http://www.openbsd.org/papers/csw03/index.html`> (accessed: 19.12.2011).

[2] CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') <`http://cwe.mitre.org/top25/#CWE-120`> (accessed 13.12.2011).

[3] Evans D. and Larochelle D., Improving Security Using Extensible Lightweight Static Analysis, In: *IEEE Software*, Jan/Feb 2002.

[4] Grünbacher A., POSIX Access Control Lists on Linux. <`http://www.suse.de/~agruen/acl/linux-acls/online/`> (accessed 19.11.2011).

[5] Harrison M. A., Ruzzo W. L., Ullman J. D., Editor R. S. Gaines *Protection in operating systems*,

[6] IBM Research. 2001. GCC extension for protecting applications from stack-smashing attacks. <`http://www.trl.ibm.com/projects/security/ssp/`> (referenced 10.11.2011).

[7] malloc, calloc, realloc, free, cfree - memory allocation and deallocation. <`http://www.openbsd.org/cgi-bin/man.cgi?query=malloc&apropos=0&sektion=0&manpath=OpenBSD+Current&arch=i386&format=html`> (accessed 15.12.2011).

[8] Mastropaolo M., Buffer overflow attacks bypassing DEP (NX/XD bits) - part 2 : Code injection. <`http://www.mastropaolo.com/2005/06/05/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-2-code-injection/`>. Accessed 14.12.2011.

[9] Miller T. C. and de Raadt T. strlcpy and strlcat - Consistent, Safe,String Copy and Concatenation. In: *USENIX conference -99*. Monterey, California. <`http://openbsd.org/papers/strlcpy-paper.pdf`> (accessed 02.10.2011)

[10] Poll E. 2011. Software security lecture notes. The Kerckhoffs institute, Radboud. Nijmegen.

[11] Provos N., Fiedl M., Honeyman P. Preventing Privilege Escalation. In: *Proceedings of the 12th USENIX Security Symposium.* Washington, DC, 2003.

[12] Puffy at work - Code right and secure, The OpenBDS way. <`http://quigon.bsws.de/papers/2010/bsdcan/mgp00046.html`> (accessed 14.11.2011)

[13] de Raadt T., Discussion on buffer overflow prevention issues. <`http://openbsd.org/papers/csw03/mgp00015.html`> (accessed 12.12.2011).

[14] Splint - Annotation-Assisted Lightweight Static Checking. <`http://www.splint.org`> (accessed: 02.12.2011).

[15] TCP Wrappers <`http://www.softpanorama.org/Net/Network_security/TCP_wrappers/index.shtml`> (accessed 18.10.2011).

[16] The open group base specifications issue 6. <`http://pubs.opengroup.org/onlinepubs/009604599/functions/strncat.html`> (accessed: 11.12.2011).

[17] Tremblay J-P and Sorenson P. G., *Theory and Practice of Compiler Writing*, Global Media, Hyderabad, India, 2008.

[18] Wilander J. and Kamkar M., A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, Linköpings universitet

[19] OpenBSD operating system. <`http://www.openbsd.org`> (accessed: 04.11.2011).

[20] OpenSSH <`http://www.openssh.org`> (accessed: 04.10.2011).

[21] WU-FTPD Development Group. <`http://wu-ftpd.therockgarden.ca`> (accessed: 02.12.2011).

[22] Zannone N. 2010-2011. Distributed trust management lecture notes. Department of Mathematics and Computer Science. Eindhoven University of Technology.

5 APPENDIX I

Stack trace regarding the inappropriately configured system.

```
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: You have an error in your
SQL syntax; check the manual that corresponds to your MySQL server version for
the right syntax to use near 'and afdeling.provincie in (0,3) order by
afdeling.sort' at line 3
        at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:936)
        at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2941)
        at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
        at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)
        at com.mysql.jdbc.Connection.execSQL(Connection.java:3243)
        at com.mysql.jdbc.Connection.execSQL(Connection.java:3172)
        at com.mysql.jdbc.Statement.execute(Statement.java:706)
        at com.mysql.jdbc.Statement.execute(Statement.java:783)
        at coldfusion.server.j2ee.sql.JRunStatement.execute(JRunStatement.java:348)
        at coldfusion.sql.Executive.executeQuery(Executive.java:1210)
        at coldfusion.sql.Executive.executeQuery(Executive.java:1008)
        at coldfusion.sql.Executive.executeQuery(Executive.java:939)
        at coldfusion.sql.SqlImpl.execute(SqlImpl.java:325)
        at coldfusion.tagext.sql.QueryTag.executeQuery(QueryTag.java:831)
        at coldfusion.tagext.sql.QueryTag.doEndTag(QueryTag.java:521)
        at cfindex2ecfm893406114.runPage(D:\websites\kicker\content\Uitslagen\index.cfm:32)
        at coldfusion.runtime.CfJspPage.invoke(CfJspPage.java:192)
        at coldfusion.tagext.lang.IncludeTag.doStartTag(IncludeTag.java:366)
        at coldfusion.filter.CfincludeFilter.invoke(CfincludeFilter.java:65)
        at coldfusion.filter.ApplicationFilter.invoke(ApplicationFilter.java:279)
        at coldfusion.filter.RequestMonitorFilter.invoke(RequestMonitorFilter.java:48)
        at coldfusion.filter.MonitoringFilter.invoke(MonitoringFilter.java:40)
        at coldfusion.filter.PathFilter.invoke(PathFilter.java:86)
        at coldfusion.filter.ExceptionFilter.invoke(ExceptionFilter.java:70)
        at coldfusion.filter.ClientScopePersistenceFilter.invoke(ClientScopePersistenceFilter.java:28)
        at coldfusion.filter.BrowserFilter.invoke(BrowserFilter.java:38)
        at coldfusion.filter.NoCacheFilter.invoke(NoCacheFilter.java:46)
        at coldfusion.filter.GlobalsFilter.invoke(GlobalsFilter.java:38)
        at coldfusion.filter.DatasourceFilter.invoke(DatasourceFilter.java:22)
        at coldfusion.CfmServlet.service(CfmServlet.java:175)
        at coldfusion.bootstrap.BootstrapServlet.service(BootstrapServlet.java:89)
        at jrun.servlet.FilterChain.doFilter(FilterChain.java:86)
        at coldfusion.monitor.event.MonitoringServletFilter.doFilter(MonitoringServletFilter.java:42)
        at coldfusion.bootstrap.BootstrapFilter.doFilter(BootstrapFilter.java:46)
        at jrun.servlet.FilterChain.doFilter(FilterChain.java:94)
        at jrun.servlet.FilterChain.service(FilterChain.java:101)
        at jrun.servlet.ServletInvoker.invoke(ServletInvoker.java:106)
        at jrun.servlet.JRunInvokerChain.invokeNext(JRunInvokerChain.java:42)
        at jrun.servlet.JRunRequestDispatcher.invoke(JRunRequestDispatcher.java:284)
        at jrun.servlet.ServletEngineService.dispatch(ServletEngineService.java:543)
        at jrun.servlet.jrpp.JRunProxyService.invokeRunnable(JRunProxyService.java:203)
        at jrunx.scheduler.ThreadPool$DownstreamMetrics.invokeRunnable(ThreadPool.java:320)
        at jrunx.scheduler.ThreadPool$ThreadThrottle.invokeRunnable(ThreadPool.java:428)
        at jrunx.scheduler.ThreadPool$UpstreamMetrics.invokeRunnable(ThreadPool.java:266)
        at jrunx.scheduler.WorkerThread.run(WorkerThread.java:66)
```